Best Available Copy

Title: System, Method and Memory Medium for Detecting Differences Between Graphical Programs

Inventor: Ray Hsu

Priority Data

This application is a continuation of U.S. Patent Application Serial No. 08/870,262 titled "Method for Detecting Differences Between Graphical Programs", filed June 6, 1997 whose inventor was Ray Hsu.

Microfiche Appendix

The present application includes a microfiche source code appendix comprising a portion of a C language source code listing of one embodiment of the software of the present invention. The appendix comprises 3 microfiche and a total of 201 frames. The appendix is comprised in U.S. Patent No. 5,974,254 titled "Method for Detecting Differences Between Graphical Programs" which was filed June 6, 1997 as U.S. Patent Application Serial No. 08/870,262, the above patent and appendix being hereby incorporated by reference as though fully and completely set forth herein.

Field of the Invention

The present invention relates to graphical programming, and in particular to detecting differences between graphical programs.

Description of the Related Art

Traditionally, high level text-based programming languages have typically been used by programmers in writing applications programs. Many different high level programming languages exist, including BASIC, C, FORTRAN, Pascal, COBOL, ADA, APL, etc. Programs written in these high level languages are translated to the machine language level by translators known as compilers. The high level text-based programming languages in this level, as well as the assembly language level, are referred

Best Available Copy

to in this disclosure as text-based programming environments.

Increasingly computers are required to be used and programmed by those who are not highly trained in computer programming techniques. When traditional text-based programming environments are used, the user's programming skills and ability to interact with the computer system often become a limiting factor in the achievement of optimal utilization of the computer system.

There are numerous subtle complexities which a user must master before he can efficiently program a computer system in a text-based environment. In addition, the task of programming a computer system to model a process often is further complicated by the fact that a sequence of mathematical formulas, mathematical steps or other procedures customarily used to conceptually model a process often does not closely correspond to the traditional text-based programming techniques used to program a computer system to model such a process. For example, a computer programmer typically develops a conceptual model for a physical system which can be partitioned into functional blocks, each of which corresponds to actual systems or subsystems. Computer systems, however, ordinarily do not actually compute in accordance with such conceptualized functional blocks. Instead, they often utilize calls to various subroutines and the retrieval of data from different memory storage locations to implement a procedure which could be conceptualized by a user in terms of a functional block. In other words, the requirement that a user program in a text-based programming environment places a level of abstraction between the user's conceptualization of the solution and the implementation of a method that accomplishes this solution in a computer program. Thus, a user often must substantially master different skills in order to both conceptually model a system and then to program a computer to model that system. Since a user often is not fully proficient in techniques for programming a computer system in a text-based environment to implement his model, the efficiency with which the computer system can be utilized to perform such modeling often is reduced.

An example of a field in which computer systems are employed to model physical systems is the field of instrumentation. An instrument is a device which collects information from an environment and displays this information to a user. Examples of

various types of instruments include oscilloscopes, digital multimeters, pressure sensors, etc. Types of information which might be collected by respective instruments include: voltage, resistance, distance, velocity, pressure, frequency of oscillation, humidity or temperature, among others. An instrumentation system ordinarily controls its constituent instruments from which it acquires data which it analyzes, stores and presents to a user of the system.

Computer control of instrumentation has become increasingly desirable in view of the increasing complexity and variety of instruments available for use. However, when first introduced, computer-controlled instrumentation systems had significant drawbacks. For example, due to the wide variety of possible testing situations and environments, and also the wide array of instruments available, it was often necessary for a user to develop a program to control the new instrumentation system desired. As discussed above, computer programs used to control such improved instrumentation systems had to be written in conventional text-based programming languages such as, for example, assembly language, C, FORTRAN, BASIC, or Pascal. Traditional users of instrumentation systems, however, often were not highly trained in programming techniques and, in addition, traditional text-based programming languages were not sufficiently intuitive to allow users to use these languages without training. Therefore, implementation of such systems frequently required the involvement of a programmer to write software for control and analysis of instrumentation data. Thus, development and maintenance of the software elements in these instrumentation systems often proved to be difficult.

U.S. Patent Nos. 4,901,221; 4,914,568; 5,291,587; 5,301,301; and 5,301,336; among others, to Kodosky et al disclose a graphical system and method for modeling a process, i.e., a graphical programming environment which enables a user to easily and intuitively model a process. The graphical programming environment disclosed in Kodosky et al can be considered the highest and most intuitive way in which to interact with a computer. A graphically based programming environment can be represented at a level above text-based high level programming languages such as C, Pascal, etc. The method disclosed in Kodosky et al allows a user to construct a diagram using a block

diagram editor, such that the diagram created graphically displays a procedure or method for accomplishing a certain result, such as manipulating one or more input variables to produce one or more output variables.

As the user constructs the data flow diagram using the block diagram editor, data structures are automatically constructed which characterize an execution procedure which corresponds to the displayed procedure. The graphical program may be compiled or interpreted by a computer. Therefore, a user can create a computer program solely by using a graphically based programming environment. This graphically based programming environment may be used for creating virtual instrumentation systems, industrial automation systems, modeling processes, and simulation, as well as for any type of general programming.

Therefore, Kodosky et al teaches a graphical programming environment wherein a user places or manipulates icons in a block diagram using a block diagram editor to create a data flow "program." A graphical program for controlling or modeling devices, such as instruments, processes or industrial automation hardware, is referred to as a virtual instrument (VI). In creating a virtual instrument, a user may create a front panel or user interface panel. The front panel includes various front panel objects, such as controls or indicators that represent or display the respective input and output that will be used by the graphical program or VI, and may include other icons which represent devices being controlled. When the controls and indicators are created in the front panel, corresponding icons or terminals may be automatically created in the block diagram by the block diagram editor. Alternatively, the user can place terminal icons in the block diagram which may later cause the display of corresponding front panel objects in the front panel, either at edit time or at run time.

During creation of the graphical program, the user selects various functions that accomplish his desired result and connects the function icons together. For example, the functions may be connected in a data flow and/or control flow format. The functions may be connected between the terminals of the respective controls and indicators. Thus the user creates or assembles a data flow program, referred to as a block diagram, representing the graphical data flow which accomplishes his desired function. The

assembled graphical program may then be compiled or interpreted to produce machine language that accomplishes the desired method or process as shown in the block diagram.

A user may input data to a virtual instrument using front panel controls. This input data propagates through the data flow block diagram or graphical program and appears as changes on the output indicators. In an instrumentation application, the front panel can be analogized to the front panel of an instrument. In an industrial automation application the front panel can be analogized to the MMI (Man Machine Interface) of a device. The user may adjust the controls on the front panel to affect the input and view the output on the respective indicators. Alternatively, the front panel may be used merely to view the input and output, and the input may not be interactively manipulable by the user during program execution.

Thus, graphical programming has become a powerful tool available to programmers. Graphical programming environments such as the National Instruments LabVIEW product have become very popular. Tools such as LabVIEW have greatly increased the productivity of programmers and more and more programmers are using graphical programming environments to develop their software applications. In particular, graphical programming tools are being used more frequently for large projects. However, large projects in which multiple software developers are working together often introduces many problems.

One of the problems often encountered when working on large software development projects is keeping track of changes to software programs. With one or two developers working on a project, disciplined efforts to document the changes to the programs may suffice. However, with more developers making changes to the same set of programs, project management becomes exceedingly difficult. In addition, an individual developer may desire to detect changes which he or she has made between two different versions of a program, such as to determine why a bug was introduced or to examine changes made to a template version of the program.

This problem has been solved somewhat satisfactorily for software programs developed in text-based languages. As previously described, text-based software programs are software programs which are written in traditional, i.e., non-graphical,

programs. An example of such a utility for text-based programs is the "diff" utility commonly found in various versions of the UNIX® operating system.

While a utility such as "diff" may require skill to develop, the basic concept of parsing through the characters of text files and comparing their differences is relatively straightforward. However, the problem of keeping track of changes in graphical programs is a much more difficult problem since graphical programs comprise graphical representations of programming constructs. As previously mentioned, graphical programs typically comprise functional blocks graphically represented as icons, not as text. A graphical program also may include control/indicator icons for providing and/or displaying data input to and output from the graphical program. Thus far, no solutions to this problem have been provided with respect to graphical programs. Therefore, a system and method for detecting differences between different versions of a graphical program are desired.

textual-based programming languages such as FORTRAN, Pascal and the C language. In particular, one characteristic of text-based programs is that they are written as text files. That is, typically, the source code of the text-based software program is comprised within a text file, such as an ASCII character set text file. Thus, the constructs such as blocks, functions, statements, operations, etc. of text-based software programs are represented as words or numbers or combinations thereof.

Software utilities have been written to detect changes between two different versions of a text-based software program. The utilities typically take as input two source code text files, each containing a different version of the program, and parse through the characters of the text files comparing each character to detect the differences in the characters between the two text files. The differences in characters are then displayed so that the user can readily discern the difference between the characters of the two programs. An example of such a utility for text-based programs is the "diff" utility commonly found in various versions of the UNIX® operating system.

While a utility such as "diff" may require skill to develop, the basic concept of parsing through the characters of text files and comparing their differences is relatively straightforward. However, the problem of keeping track of changes in graphical programs is a much more difficult problem since graphical programs comprise graphical representations of programming constructs. As previously mentioned, graphical programs typically comprise functional blocks graphically represented as icons, not as text. A graphical program also may include control/indicator icons for providing and/or displaying data input to and output from the graphical program. Thus far, no solutions to this problem have been provided with respect to graphical programs. Therefore, a system and method for detecting differences between different versions of a graphical program are desired.

Summary of the Invention

The present invention provides a method for detecting differences between two graphical programs. Preferably, the method is executed on a computer system including a display screen and an input device. In one embodiment, the graphical programs are used to control instruments or other devices coupled to the computer system. The first graphical program comprises a first plurality of objects and the second graphical program comprises a second plurality of objects. The objects include attributes and methods and have a specific object type. Preferably, the objects are represented visually as icons in a user interface panel and/or block diagram. The user interface panel is used to provide input to and receive output from the graphical program. The block diagram comprises graphical code including user interface panel terminals and function blocks connected by data flow paths, or signals, to perform the desired function of the graphical program. The block diagram receives input for the graphical code from the user interface panel and provides output of the graphical code to the user interface panel. The graphical programs also include attributes themselves.

The method comprises first creating data structures to represent the first and second graphical programs. Preferably, the data structures include a directed graph for each of the block diagrams and user interface panels of each of the graphical programs. The vertices of the graphs correspond to the objects of the block diagram or user interface panel. The edges of the block diagram graph correspond to the data flow paths. The edges of the user interface panel graph indicate the hierarchical relationships of the user interface panel objects, such as parent/child relationships in data aggregation clusters.

The method further comprises a step of matching the first plurality of objects of the first graphical program with the second plurality of objects of the second graphical program. The method attempts to match objects of the graphical programs to determine similarities between the two programs, and hence to aid in finding differences between them. Preferably, the matching is performed according to a matching heuristic which calculates scores indicating a degree of similarity between an object in the first graphical program and an object in the second graphical program according to one or more criteria. These scores, or matching information, are stored in a match matrix data structure. The

rows of the match matrix correspond to the objects of the first graphical program and the columns of the match matrix correspond to the objects of the second graphical program. The matching is performed for both the block diagram graphs and the user interface panel graphs.

Pairs of objects (i.e., one from each of the graphical programs) are scored which match according to object type and which have no conflicts with other objects according to object type. Using these matching objects as starting points, the graphs are traversed looking for matching edges. The score of the source and destination objects of matching edges are updated accordingly. The matrix is then resolved by selecting the highest score in a given row or column. That is, in a given row, for example, the object corresponding to the column element in the match matrix with the highest score is selected to match the object corresponding to the given row. The other elements in the row are then zeroed out to indicate the resolved match. If a tie in scores exists at this point, the tie is not resolved, but postponed until a later step. Thus, when the match matrix is eventually resolved with all ties broken, a 1:1 or 1:0 relationship exists between objects in the first graphical program and objects in the second graphical program, the relationship depending upon whether or not an object in the first graphical program has a matching object in the second graphical program.

After the matrix is resolved as described above, yet unmatched objects are scored according to object type only, and the match matrix is resolved again where possible. Next, any remaining conflicts are scored by examining the immediately neighboring objects of the graphs and the match matrix is resolved again where possible. Next, any remaining conflicts are scored by object attributes using a compare engine which includes specific knowledge of the attributes of each of the object types and the match matrix is resolved. Finally, the remaining conflicts are scored by the positions and sizes of the objects. After this step, the matrix is resolved and tie scores are resolved.

The method then determines differences between the first graphical program and the second graphical program in response to the matching step. First, objects which match exactly are grouped into lists of matching subgraphs. Objects match exactly if they are of the same object type, their attributes match, and they have matching edges.

09520904.093399

The remaining objects which do not match exactly are then grouped into lists of non-exact matching subgraphs. The lists of non-exact matching subgraphs are matched using a subgraph match matrix in a similar manner to which the objects themselves were matched using a match matrix. The non-exact matching subgraphs in the subgraph match matrix are scored based upon matching nodes in the original match matrix. The rows and columns of the subgraph match matrix correspond to the subgraphs in the non-exact matching subgraph lists. Considering direct and transitive connections, the non-exact matching subgraphs are merged into composite non-exact matching subgraph lists. The compare engine is then used to compare the individual objects in the non-exact matching subgraphs to determine additional differences. Furthermore, non-exact matching subgraphs, which may be single objects, which have no match in the match matrix are stored as differences. For the user interface panels, top-level objects are compared using the compare engine and then the low-level objects, e.g., the data elements of any clusters, are compared using the compare engine to determine differences. All of the differences determined are stored in a results data structure.

The method further comprises displaying an indication of the stored differences on the display screen of the computer system. In one embodiment, the differences are highlighted by greying out the portions of the block diagram which are not part of the difference and a geometric symbol, such as an ellipse or circle, is drawn around the difference. In one embodiment, the differences are displayed in a distinguishing color. In one embodiment, the differences are surrounded with "marching ants". In one embodiment, displaying the differences includes displaying a text description describing the differences.

Brief Description of the Drawings

A better understanding of the present invention can be obtained when the following detailed description of the preferred embodiment is considered in conjunction with the following drawings, in which:

Figure 1 is an illustration of an instrumentation control system;

Figure 2 is a screen shot of an example of a virtual instrument or graphical software program for controlling an instrumentation control system such as in Figure 1;

Figure 3 is a flowchart illustrating the method of detecting differences between graphical programs, such as the graphical program of Figure 2, according to the present invention;

Figure 4 is a screen shot of the method of the present invention displaying differences between two graphical programs;

Figure 5 is a screen shot of a dialog which allows a user to select various options for detecting differences between two graphical programs;

Figure 6 is a block diagram illustrating a graph data structure according to a preferred embodiment of the present invention;

Figure 7 is a matrix list according to a preferred embodiment for storing an n row by m column matrix;

Figure 8 is a flowchart illustrating in more detail the step of matching objects of two graphical programs from Figure 3;

Figure 9 is a block diagram illustrating a match matrix according to a preferred embodiment of the present invention;

Figure 10 is a flowchart illustrating in more detail the step of determining differences between block diagrams of two graphical programs of Figure 3;

Figure 11 is a flowchart illustrating in more detail the step of determining differences between the user interface panels of two graphical programs of Figure 3.

While the invention is susceptible to various modifications and alternative forms specific embodiments are shown by way of example in the drawings and will herein be described in detail. It should be understood however, that drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed.

But on the contrary the invention is to cover all modifications, equivalents and alternative following within the spirit and scope of the present invention as defined by the appended claims.

Detailed Description of the Preferred Embodiment

The present invention provides a method for detecting differences between two different graphical programs. Although in one embodiment described below the graphical programs control instrumentation hardware, it is noted that graphical programs may be used for a plethora of software applications and are not limited to instrumentation applications. Likewise, the method of the present invention is operable for use in detecting differences in graphical programs written for any application.

Figure 1 - Computer System

Referring now to Figure 1, an instrumentation control system 10 is shown. The system 10 comprises a computer 12, which connects to one or more instruments. The computer comprises a CPU, a display screen, and one or more input devices such as a mouse or keyboard as shown.

The one or more instruments may include a GPIB instrument 14, a VXI instrument 16, a serial instrument 18 and/or a data acquisition board 20. The GPIB instrument 14 is coupled to the computer 12 via a GPIB interface provided by the computer 12. The VXI instrument 16 is coupled to the computer 12 via a VXI bus or MXI bus provided by the computer. The serial instrument 18 is coupled to the computer 12 through a serial port, such as an RS-232 port, provided by the computer 12. Finally, the data acquisition board 20 is coupled to the computer 12, typically by being plugged in to an I/O slot in the computer such as a PCI bus slot, an ISA bus slot, an EISA bus slot, or a MicroChannel bus slot provided by the computer 12. In typical instrumentation control systems an instrument will not be present of each interface type and in fact many systems may only have one or more instruments of a single interface type, such as only GPIB instruments.

The instruments are coupled to a unit under test (UUT) 23, process or are coupled to receive field signals, typically generated by transducers. The system 10 may be used in a data acquisition and control application, or may instead be used in a test and measurement application. If the system 10 is used in a data acquisition application, the system 10 may also include signal conditioning circuitry 21 coupled between the data acquisition board 20 and transducers.

The instruments are controlled by graphical software programs which are stored in memory of the computer and which execute on the computer 12. The graphical software programs which perform instrumentation control are also referred to as virtual instruments. Figure 2 illustrates an example of a virtual instrument or graphical software program. In the embodiment described herein, the graphical programs used herein were created using the LabVIEW graphical programming system.

Referring again to Figure 1, the system 10 preferably includes a memory media, such as a non-volatile memory, e.g., magnetic media, a system memory, CD-ROM, or floppy disks 22, on which computer programs according to the present invention are stored. The present invention comprises a software program stored on a memory and/or hard drive of the computer 12 and executed by a CPU of the computer. The CPU executing code and data from the memory thus comprises a means for detecting differences in graphical programs according to the steps described below.

Figure 3 - High-level Diff Flowchart

Referring now to Figure 3, a flowchart illustrating the method of detecting differences between graphical programs according to the present invention is shown. Preferably, the method of the present invention is embodied as a software program which executes on a computer system such as computer system 12 of Figure 1. The software program of the present invention for detecting differences between graphical programs will subsequently be referred to as "diff" for brevity. Appendix A includes a portion of a C language source code listing of one embodiment of diff.

In the preferred embodiment, the graphical programs use graphical data flow programming, such as the LabVIEW graphical programming environment. However, other graphical programming systems may employ the method described herein to detect differences between graphical programs. Examples of systems which may employ the method are Visual Designer from Intelligent Instrumentation, Hewlett-Packard's VEE (Visual Engineering Environment), Snap-Master by HEM Data Corporation, DASYLab by DasyTec, and GFS DiaDem, among others. Programming environments which

include graphical programming elements can also use the graphical diff method of the present invention.

Diff receives as input two graphical programs, such as the graphical program of Figure 2, in step 100. Each of the graphical programs includes a plurality of objects. An object may be defined as having attributes, or properties, and methods according to the notion of objects in the art of object oriented programming. Preferably, an object has an associated icon which visually represents the object, as shown in Figure 2. Preferably, the graphical program comprises a block diagram portion and a user interface panel portion, and the objects are arranged in these two portions: the block diagram portion and the user interface panel portion. Alternatively, the objects are comprised solely in a block diagram or graphical program portion. In this embodiment, user interface objects, if present, may be comprised in the block diagram portion. A user interface panel is shown in the window in the upper right hand portion of Figure 2 and a block diagram is shown in the window in the lower left hand portion of Figure 2. In the case of instrumentation control applications, the user interface panel is typically referred to as an instrument front panel, or front panel. The objects in the user interface panel include controls and indicators. Controls are used to receive input, typically from a user, and to provide the input data to the block diagram. Indicators are used to receive output data from the block diagram and display the output data to the user. Examples of indicators are graphs, thermometers, meters and gauges. Examples of controls are slides, knobs, switches, buttons and dials. In one embodiment, the controls and indicators include clusters of controls and indicators arranged in a hierarchical manner.

Preferably, the user interface panel comprises a directed acyclic graph of objects. In particular, the user interface panel comprises a hierarchical tree structure wherein the nodes of the tree are the user interface panel objects. A graph may be defined as a finite non-empty set of vertices and a set of edges such that every edge connects exactly two vertices. Preferably, any two vertices may be connected by zero or more edges. In the user interface panel, the vertices are the control and indicator objects and the edges are the hierarchical relationship between the objects. The direction of the edges are determined by the level in the hierarchy. That is, the direction is from higher level

objects to lower level objects. The connectivity of an object is related to the other objects to which it is connected, i.e., its neighboring objects, and the edges by which it is connected to its neighboring objects.

The block diagram is the portion of the graphical program which includes the graphical code to perform the calculations and operations of the graphical program application. The objects in the block diagram include terminals associated with the front panel controls and indicators. The front panel terminals are used to input and output data between the front panel controls/indicators and the function blocks of the block diagram. The block diagram objects also include function nodes, such as mathematical operators; code execution structures such as for loops, while loops, case statements, and variable references; string functions; file I/O nodes; communication nodes; instrument I/O nodes; and data acquisition nodes, for example. Preferably, the block diagram nodes are connected by data paths, or signals, which determine the flow of data through the block diagram.

In the preferred embodiment, the block diagram comprises a data flow diagram arranged as a directed acyclic graph. The vertices of the graph are the terminals and nodes of the block diagram. The edges of the graph are data path signals which connect the nodes and terminals. The nodes themselves comprise one or more terminals which are connected to the edges. The direction of the edges of the graph is determined by the nodes themselves. For example, if a signal is connected between an output terminal of a first node and an input terminal of a second node, then the direction of data flow on that edge is from the output terminal to the input terminal.

In response to receiving the two graphical programs, diff creates a data structure representing the first block diagram, a data structure representing the second block diagram, a data structure representing the first user interface panel, and a data structure representing the second user interface panel, in step 102. Preferably, the data structures comprise directed graphs, and more particularly, directed acyclic graphs. The graphs are used by diff to determine differences between the block diagrams and user interface panels of the two graphical programs. Step 102 and the graph structure will be discussed in more detail with regard to Figure 6.

Diff then matches objects in the first graphical program with objects in the second graphical program, in step 104. Objects are matched according to one or more criteria, such as object type, connectivity, attributes and position. Preferably, the matching is performed by calculating a weighted score which indicates a degree of matching or similarity between an object in the first graphical program and an object in the second graphical program according to the one or more criteria to produce matching information. The matching information is used to group the objects into matching subgroups, or sub-graphs, and non-matching sub-graphs for the purpose of determining differences between the two graphical programs. The matching of objects performed in step 104 will be described in more detail with regard to Figure 8.

Diff then determines differences between the first graphical program and the second graphical program, in steps 106, 108 and 110. Differences are determined for the block diagrams, the user interface panels and the attributes of the first and second graphical programs. The determining of differences includes comparing objects found to match in step 104 to determine any differences between the matching objects. The determining of differences also includes determining objects in the first graphical program which have no match, i.e., which do not exist in the second graphical program. The differences may be functional differences or cosmetic differences. A functional difference is one which may potentially affect the execution of the graphical program. Cosmetic differences are differences which do not affect execution of the graphical program. If the same set of inputs to a graphical program produce the same set of outputs even though a change has been made, that difference is a cosmetic difference rather than a functional difference. The determining of differences in the block diagrams and user interface panels performed in steps 106 and 108, respectively, will be described in more detail below.

Preferably, the graphical program also includes attributes. Examples of graphical program attributes include an icon and connector representing the virtual instrument; attributes related to execution of the graphical program, such as execution priority and whether or not to run the graphical program upon being loaded; attributes related to the visual presentation of the graphical program, such as whether or not to display toolbars

and window scroll bars; documentation-related attributes; attributes relating the history of the graphical program; and selection of a run-time menu. Diff detects differences between the attributes of the two graphical programs, in step 110.

Once diff determines the differences between the two graphical programs, diff displays on the display screen of the computer system 12, an indication of the differences, in step 112. In one embodiment, diff highlights differences by drawing a geometric symbol, such as an ellipse or circle, around the differences, as shown in Figure 4. Figure 4 shows two versions of a virtual instrument named "Calculate Num Iterations" and "Calculate Num Iterations2". The block diagrams of the two graphical programs are displayed side by side in Figure 4. One of the differences between the two graphical programs is shown in Figure 4. The difference is related to a constant node which is connected as an input to a Select node. The block diagram on the left has a constant node with a value attribute of 10, whereas, the block diagram on the right has a constant node with a value attribute of 1. Figure 4 shows the constant node in each block diagram with an ellipse drawn around them to highlight the difference. It is noted that the portions of the block diagram which are not part of the currently selected difference are "greyed out" so that the difference may be visually highlighted for the user.

In operation of one embodiment, diff also displays the differences in a distinguishing color. For example, a black rectangular background is displayed behind the constant node in the first block diagram for a first period of time (such as two seconds), and then a black rectangular background is displayed behind the constant node in the second block diagram for a similar period of time. This highlights the difference for the user so that the user can visually distinguish the difference.

In operation of another embodiment, diff surrounds the differences with "marching ants". Marching ants refers to the visual movement of alternating colors, such as black and white, around a differing object in each of the graphical programs. This operation may be partially seen in Figure 4 by virtue of the fact that the constant node and the Select node are circumscribed with an alternating black and white portion thick line. Likewise, the wire connecting the two nodes is highlighted in a similar manner. In

operation, the alternating black and white portions "move" visually around the nodes and along the wire to highlight the difference.

Displaying the differences may also include displaying a text description of the differences. Figure 4 shows a "Differences" window 300 at the bottom portion of the screen. The left portion 302 of the Differences window lists five differences between the two block diagrams shown. The right portion 304 of the Differences window provides a more detailed textual description of the difference highlighted in the left portion and which is currently displayed in the block diagrams, namely, a numeric constant data value. In Figure 4, the text description reads "Numeric Constant: data value" to indicate that there is a difference between the value of 10 and the value of 1 in the two constant nodes highlighted.

Preferably, differences may be determined in the block diagram, front panel and/or graphical program attributes individually or in any combination thereof, rather than determining the differences in all three. Figure 5 illustrates a screen shot of a dialog which allows the user to select various diff options.

Figures 6 and 7 - Data Structures

Referring now to Figure 6, a block diagram illustrating a graph data structure according to a preferred embodiment is shown. As described above, a graph data structure is created in step 102 to represent each of the block diagram and front panel of each of the two graphical programs. Preferably, a graph structure is a data structure which represents a directed graph. In the case of the block diagram, the vertices of the graph are the user interface panel terminals and the node terminals of the block diagram. The edges of the graph are data path signals which connect the terminals. That is, each edge in the graph includes a source terminal and a destination terminal. The direction of data flow is from the source terminal to the destination terminal.

In the case of the user interface panel, the vertices of the graph are the user interface panel objects, i.e., the controls and indicators. The direction of the edges are determined by the level in the hierarchy. That is, the direction is from higher level objects to lower level objects. The hierarchy of objects in the user interface panel with

relation to array and cluster controls and indicators will be described in more detail below with reference to Figure 11.

The vertices of the graph are stored as a vertex list 324, or object list, as shown. The number of elements in the object list is equal to the number of objects in the graph. Each object list element includes an object ID field and a flags field.

In the preferred embodiment, the graph data structure is represented as an adjacency matrix 322, as shown in Figure 6. The rows of the matrix correspond to the objects of the graph, i.e., the block diagram or user interface panel. One row exists for each object in the graph. Likewise, the columns of the matrix represent the objects of the graph, one for each object. Thus, a matrix element exists for each ordered pair defined by a row and column (i,j). If an edge exists between the two objects in the ordered pair, then a non-empty edge list 326 exists in the matrix. If no edges exist between the two objects in the ordered pair, then the edge list is empty.

Figure 6 also illustrates a typical edge list. An element exists in the edge list for each edge between the two objects of the respective row and column. Each element in the edge list comprises a source terminal identifier field, a destination terminal identifier field and a flags field. Data flows along the edge from the source terminal to the destination terminal. The flags are used in the steps of matching objects and determining differences. The source terminal and destination terminal fields are used to reference the objects in the object list.

The adjacency matrix representation of the graph structure is typically a sparse matrix. That is, typically, a given object is connected to a relatively small number of other objects in the graph. Thus, in a given row, for example, the edge list will be empty for most of the columns. In one embodiment the adjacency matrix is stored as a matrix list. A matrix list embodiment advantageously makes more efficient use of memory storage for relatively sparse, non-trivial graphs than a full matrix representation.

Referring now to Figure 7, a matrix list according to a preferred embodiment is shown for storing an n x m matrix, i.e., a matrix with n rows and m columns. A matrix list may be used to store a matrix of any dimensions. That is, the matrix list is not limited to representing an n x n matrix, but rather may be used to represent an n x m matrix. The

matrix list comprises a rowList, a colList and an eltList, as shown. The eltList, or element list, is a list of the elements of an adjacency matrix. With regard to the graph of Figure 6, the eltList elements would be the edges in all of the edge lists. The eltList may include a maximum of n x m elements.

The rowList, or row list, comprises an array of list pointers indexed by row number. Each row list element includes a column number and eltIndex, or element index. The column number in the row list element is combined with the row number used to index the row list array to produce an ordered pair corresponding to the (i,j) element in the graph adjacency matrix. The eltIndex is used to index into the eltList array to select the desired matrix element, as shown, such as an edge of the graph of Figure 6.

Similarly, the colList, or column list, comprises an array of list pointers indexed by column number. Each column list element includes a row number and eltIndex. The row number in the column list element is combined with the column number used to index the column list array to produce an ordered pair corresponding to the (i,j) element in the graph adjacency matrix.

Figure 8 - Matching Objects Flowchart

Referring now to Figure 8, a flowchart illustrating in more detail step 104 from Figure 3 of matching objects of the two graphical programs is shown. Preferably, step 104, i.e., steps 120 through 132 are performed once for the block diagrams and once for the user interface panels. For brevity, the steps of Figure 8 will be described with reference to the block diagram. First, diff creates a match matrix to match elements between the first block diagram and the second block diagram, in step 120. The match matrix is used to determine similarities between the two block diagrams so that differences may subsequently be determined. A match matrix is also created to match elements between the first and second user interface panels in step 120. As mentioned, for brevity, the steps of Figure 8 will be understood to also be performed for the user interface panels as well as the block diagrams, although the steps are described with reference to the block diagrams.

Referring now to Figure 9, a block diagram illustrating a match matrix according to a preferred embodiment of the present invention is shown. The match matrix comprises a matrix of match information elements. The rows of the match matrix correspond to the objects of the first graphical program and the columns of the match matrix correspond to the objects of the second graphical program. Figure 9 illustrates a match matrix wherein the first graphical program includes n objects and the second graphical program includes m objects. That is, the match matrix is an n row by m column matrix. Thus, a given match matrix element (i,j) includes matching information regarding an object pair, wherein the objects of the pair are the ith object of the first graphical program and the jth object of the second graphical program.

The match information includes a score field and a flags field. The score field indicates a degree of matching between the two objects of the object pair. The object pairs are scored using a matching heuristic. Objects may match according to various criteria, such as object type and associated conflict in object type, connectivity, neighboring objects, object attributes, position and size. Each of these matching criteria are given a different weight according to the matching heuristic. Scoring of the object pairs will be described in more detail below. The flags field is used in the process of matching objects and determining differences. In the preferred embodiment, the match matrix is embodied as a matrix list as shown in Figure 7, wherein the elements in the element list are match information elements.

Referring again to Figure 8, diff calculates scores for object pairs wherein the object type, or object id, of an object in the first graphical program matches the object type of an object in the second graphical program and wherein there are no other objects in the two graphical programs with the same object type, in step 122. For example, if there is only one add node in each of the two graphical programs, then the add nodes match according to object type and are without conflicts. When two objects match according to object type and are without conflicts, diff assigns a relatively large initial score, or weight, to the object pair, thereby effecting a high probability that when the match matrix is resolved, the two objects will be matched.

Using the matching object pairs produced by step 122, diff traverses the graphs of the graphical programs searching for matching edges and increases the score of the source and destination object connected to each matching edge found, in step 124. That is, diff scores the objects by examining the connectivity of the objects. A matching edge is an edge in which both the source and destination objects are elements of matching object pairs according to step 122. According to one embodiment of the matching heuristic, the weight given to objects attached to a matching edge is relatively large.

After scoring the match matrix elements, diff resolves the match matrix where possible. Resolving the match matrix includes recursively traversing the rows of the match matrix, i.e., the objects of the first graphical program, and selecting the element in a given row with the highest score and then setting the score of the other elements in the given row to zero. Thereby, a 1:1 correspondence is produced between the objects in the first and second graphical programs based on the match scores. That is, every object in the first graphical program can contain at most one match with an object in the second graphical program, or alternatively, a row can contain at most one matching column.

A row with all zeroes indicates that the object of the first graphical program corresponding to the row has no matching object in the second graphical program. That is, for some objects a 1:0 correspondence is produced between the objects in the first and second graphical programs based on the match scores. The same process is performed for the columns of the match matrix, i.e., for the second graphical program. At this step in the matching process, if two scores in a given row (or column) are equal, diff does not resolve the row (or column). Instead, the row scores are left intact in hopes that they will be resolved according to a different criteria in a subsequent step. That is, diff does not perform any tie breaks at this point in the matching process.

Next, diff scores unmatched objects according to object type only, in step 126. That is, diff assigns scores to objects which have the same object type. However, in this step 126 the requirement that objects match without conflict is not imposed as in step 122. Preferably, according to step 126, object pairs are assigned a score which is relatively smaller than the score assigned for matches according to step 122 or step 124. That is, the weight attributed to a match based on connectivity and/or non-conflicting

object type matches is greater than the weight attributed to a match based solely on object type. After scoring unmatched objects according to object type, diff resolves the match matrix without using tie breaks, in step 126.

Next, diff scores remaining conflicts, i.e., objects which match according to type but which still have conflicts after resolving the matrix according to step 126, by examining immediately neighboring objects of the graph, in step 128. That is, the match score is updated by inspecting each terminal of the objects for matching upstream objects, matching downstream objects, and matching non-connections, i.e., terminals with no signals connected. After scoring remaining conflicts by examining immediate neighbors, diff resolves the match matrix without using tie breaks, in step 128.

Next, diff scores remaining conflicts by object attributes, in step 130. That is, diff invokes a compare engine to compare various attributes of the objects to determines similarities between the objects. The compare engine includes a compare method which has knowledge of each object type in order to distinctly compare the attributes of the objects according to their object types. The compare engine calculates and returns a score indicating the degree of matching or similarity between the two objects being compared. The attributes are grouped, according to the matching heuristic, into functional attributes and cosmetic attributes. Examples of functional object attributes are terminal data types and constant data values. Examples of cosmetic attributes are the position, size, color, visibility, name, etc. of objects. A functional attribute is one which may potentially affect the execution of the graphical program. Cosmetic attributes are attributes which do not affect execution of the graphical program. If the same set of inputs to a graphical program produce the same set of outputs even though an attribute has been changed, that attribute is a cosmetic attribute rather than a functional attribute. Preferably, functional attributes have greater weight than cosmetic attributes. Furthermore, match scores based on examination of attributes have less weight than matches based on connectivity or matches by object type without conflicts. After scoring by object attributes, diff resolves the match matrix without using tie breaks, in step 130.

Next, diff scores remaining conflicts by object position and size, in step 132. That is, diff compares the position and size of the objects to determines similarities between

the objects. The more similar the position and/or size of two objects the greater the match score assigned to them. After scoring by object position and size, diff resolves the match matrix using tie breaks, in step 132. That is, in step 132 diff resolves all conflicts. Preferably, tie breaks in match scores are resolved on a "first-come-first serve" basis, i.e., as the match matrix is traversed, the first score encountered among equal scores wins the tie.

Figure 10 - Determining Differences in Block Diagrams Flowchart

Referring now to Figure 10, a flowchart illustrating in more detail step 106 of determining differences between the block diagrams of two graphical programs is shown. In step 140, diff groups exact matching objects into a list of exact matching subgraphs using the match information of the match matrix created according to step 104 and employing the compare engine. In an exact match the two objects: 1) have the same object type and their attributes compare exactly according to the compare engine; and, 2) have connectivity matches. The compare engine includes a compare method which has knowledge of each object type in order to distinctly compare the attributes of the objects according to their object types. The compare engine returns a predetermined value if the two objects being compared compare exactly. In a connectivity match all connections, or edges, of the objects match. That is, all objects to which the prospective exact matching objects are connected are also matching objects. The exact matching subgraphs comprise exact matching objects which are connected together. The exact matching subgraphs are also referred to as exact matching connected components.

Diff simultaneously traverses both graphs representing the block diagrams to produce the list of exact matching subgraphs. While traversing the graphs to find exact matching subgraphs, each edge belonging to a matched object is marked in the flags field of the corresponding element in the adjacency matrix data structure to denote that the edge belongs to a matched object. Matching objects which have all of their edges marked in the corresponding adjacency matrix element in both graphs are marked in the flags field of the object list (shown in Figure 6) element corresponding to the matching object.

Next, diff groups the remaining objects, i.e., the objects which were not grouped into the exact matching subgraphs, into a list of non-exact matching subgraphs, one for each block diagram, using the match matrix and list of exact matching subgraphs, in step 142. These two lists of non-exact matching subgraphs are then matched together in a match matrix, in step 144. This match matrix is a different match matrix from the one produced in step 104, and will be referred to herein as the subgraph match matrix. The subgraph match matrix is similar to the match matrix shown in Figure 9 except that the rows correspond to the subgraphs in the list of non-exact matching subgraphs of the first block diagram and the columns correspond to the subgraphs in the list of non-exact matching subgraphs of the second block diagram. The scores of the subgraph match matrix indicate a degree of matching or similarity between a subgraph in the first block diagram and a subgraph in the second block diagram. Objects in the non-exact matching subgraphs which match according to the match matrix of step 104, are referred to as anchor nodes. The anchor nodes are used to calculate the match scores in the subgraph match matrix.

Using the matching information in the subgraph match matrix, diff merges the two lists of non-exact matching subgraphs into a composite non-exact matching subgraph, in step 146. The merging is performed such that transitive connections, i.e., non-direct connections, are taken into account when creating the composite non-exact matching subgraph. The non-exact matching subgraphs are then stored in a results data structure in step 148 so that the non-exact matching subgraph differences may be subsequently displayed to the user in step 112. Remaining non-exact matching subgraphs are paired with empty graphs and are considered as new subgraphs that have been added to or removed from the second block diagram. The remaining non-exact matching subgraphs paired with empty graphs are also stored in the results data structure so that they may be subsequently displayed to the user in step 112.

Once the exact matching and non-exact matching subgraphs have been created, diff compares matching objects in the non-exact matching subgraphs to determine additional differences in the objects of the non-exact matching subgraphs, in step 150. The additional differences are stored in the results data structure, in step 152. The

compare engine is employed to compare the matching objects. The compare engine includes a compare method which has knowledge of each object type in order to distinctly compare each attribute of the object. The compare engine produces a textual description of the differences between two objects. This textual description is displayed for the user in step 112, as described above.

The differences found by the compare engine may be functional differences or cosmetic differences. A functional difference is one which may potentially affect the execution of the graphical program. Cosmetic differences are differences which do not affect execution of the graphical program. If the same set of inputs to a graphical program produce the same set of outputs even though a change has been made, that difference is a cosmetic difference rather than a functional difference.

Figure 11 - Determining Differences in User Interface Panels Flowchart

Referring now to Figure 11, a flowchart illustrating in more detail step 108 of determining differences between the user interface panels of the two graphical programs is shown. In one embodiment, user interface panel controls and indicators may include arrays and/or clusters. In one embodiment, an array is a variable-sized collection of data elements of the same type. In one embodiment, a cluster is a fixed-sized collection of data elements of mixed types. The clusters are aggregations and/or child-parent relationships of data elements, similar to a record or structure in high-level programming languages. Clusters may include a hierarchy of objects, which include other clusters, controls, and/or indicators. Thus, a cluster may include top-level objects and low-level objects in the hierarchy of objects. As described above, in step 102, diff creates a graph for each user interface panel wherein the objects of the user interface panel are the vertices or nodes in the graph and the edges of the graph represent the hierarchical relationship between the user interface panel elements, and in particular arrays and clusters.

As described above, in step 104 a match matrix including matching information is created which is resolved to produce a 1:1 or 1:0 matching relationship between objects of the first and second graphical programs. A resolved match matrix is produced for the

block diagrams and a resolved match matrix is produced for the user interface panels according to steps 120 through 132.

For top-level user interface panel objects which match in the match matrix, diff invokes the compare engine to compare the matching objects in order to determine differences between the top-level objects, in step 160.

Next, diff provides matching information to the compare engine to compare the low-level objects (e.g., individual cluster elements) of the top-level objects in order to determine differences between the low-level objects of the user interface panels, in step 162.

Then, diff determines which objects of the user interface panels have no match, in step 164. This occurs when the user has added or deleted an object between the first user interface panel and the second user interface panel. Objects which have no match are identified as those which have a 1:0 relationship in the user interface panel match matrix. The differences found in steps 160 through 164 are stored, in step 166, so that the differences may be displayed in step 112.

Although the system and method of the present invention has been described in connection with the preferred embodiment, it is not intended to be limited to the specific form set forth herein, but on the contrary, it is intended to cover such alternatives, modifications, and equivalents, as can be reasonably included within the spirit and scope of the invention as defined by the appended claims.